a string of bytes the number of which are specified by the length field. It is encoded as defined in the BER for that type.

6.4 ASN.1 EXTERNAL Type (Universal Header)

An ASN.1 EXTERNAL type is a universal header. All ASN.1 compliant protocol interpreters can extract and interpret an EXTERNAL without ambiguity. The definition of EXTERNAL is quite flexible, but that flexibility is not needed here to meet the basic objectives of a universal header.

A simplified EXTERNAL type is encoded as:

[tag= class=0,p=1,number=8] [length] [object id] [payload]

Tag and *length* fields are encoded as described above. *Object ID* provides unambiguous self identification for the header. *Payload* is a sequence of bytes that are interpreted according to the standard indicated by the object ID.

Object ID is itself an ASN.1 type with tag, length, and value fields, and is encoded as:

[tag= 0,0,6] [length] [id value]

Object ID value is a sequence of bytes that represent the hierarchical identifier for the referenced standard. ID values are assigned, registered, and administered by CCITT and ISO in the course of standards development. Or, ID assignment can be delegated to member bodies or companies or organizations (thereby, SMPTE could assume responsibility for administering a portion of the ID space.) The root prefix values are:

```
CCITT [0]
    recommendation[0]          : CCITT committees
    question[1]                : CCITT Study Groups
    administration[2]          : country PTTs (country code)
    network operator[3]        : X.121 organizations

ISO[1]
    standard[0]                : ISO standards
    registration authority[1]  : ISO authorities
    member body[2]             : member bodies (country code)
    identified organization[3] : organizations

joint ISO CCITT[2]             : assignment delegated to ANSI
```

A few prefixes are of particular note. iso.standard registers all ISO standards. ccitt.administration and iso.memberbody are assigned to sovereign bodies (identified by their international telephone country code). iso.organization is assigned to international organizations. These cover virtually all the situations under which a header identifier will need to be assigned.

An ID value is encoded as a sequence of bytes. The first two levels are encoded in the first byte -- a.b = 40*a + b, a <= 3, b <= 39. Remaining levels are encoded

as one or more bytes as needed to represent the numerical value for that level. If a value is greater than 127, it requires more than one byte; the MSB of the byte is set to indicate that the value is continued in the next byte, and so on. For example, iso.standard.jpeg is three bytes:

iso.standard.jpeg :: 1.0.10918 = [40] [128+85] [38]

Similarly, Group 3 Fax is identified as:

ccitt.recommendation.t.4 :: 0.0.20.4 = [0] [20] [4]

The extensibility of the ID value field permits up to 126 byte long IDs to specify distinct IDs numbering to ~2^882 or ~10^265.

Payload is an ASN.1 type with tag, length, and value fields, encoded as a sequence of bytes that are interpreted according to the standard indicated by the *Object ID*.

[tag= 2,0,1] [length] [payload value]

[footnote *1: At this writing, JPEG is nearing but not yet an ISO standard. Thus, though thought to be correct, the number here (1.0.10918) is not yet official.]

6.5 ASN.1 Descriptor

In addition to the EXTERNAL type (for use as header), ASN.1 defines other basic types to represent a variety of values, including: booleans, integers, reals, byte strings, character strings, universal time code, etc., and constructions of values into arbitrary data structures.

Although a distinction is drawn between header and descriptor in this report, the ASN.1 approach permits a single mechanism to serve both functions. The full benefits of ASN.1 become apparent when it is applied to the descriptor. Especially the ability to construct new types and to incorporate references to other standards. (See current ISO work on Image Interchange Format (IIF) for an example of ASN.1 use in defining descriptors.)

6.5 How ASN.1 Addresses Objectives

The following describes how an ASN.1 header/descriptor addresses the objectives stated at the beginning of this report.

Universality -- ASN.1 header/descriptor promotes and enhances universality:

- It complies with and recognizes existing standards and practices. All existing and future ISO/CCITT standards are uniquely identified by an

ASN.1 Object ID.

- It addresses the issues of sovereignty. ISO/CCITT administers and delegates assignment of Object IDs to subcommittees, member bodies (by country code), and organizations. The complexity of this task and the benefits of leveraging existing administrative structures should not be underestimated.
- It facilitates coordination among television, telecommunications, and computer industries.
- It sets a minimal level of compliance for low cost receivers. Furthermore, the advanced stage of definition, tools, and expertise will facilitate the rapid deployment of header compliant devices.

Longevity -- ASN.1 has inherent longevity:

- All fields of ASN.1 types (tag, length, value) can be extended. Payload lengths from a few bytes to ~10^303 can be represented. Similarly, Object IDs can range from a few bytes to ~10^265 bytes.
- It has a preexisting registry, and is self maintaining. ISO/CCITT already registers, administers, and delegates assignment of Object IDs.
- It defines immutable identifiers. Once an Object ID is assigned, it exists "for all time". In the future, when an old ASN.1 header is recognized, there is no ambiguity to the referenced standard.

Extensibility -- ASN.1 is inherently extensible:

- All fields of ASN.1 types can be extended without redefinition.
- New types can be defined and their encoding automatically generated without the need to introduce new rules.
- Since ASN.1 is fully defined, any compliant receiver and equipment are guaranteed to be able to recognize future extended ASN.1 headers.

Interoperability -- ASN.1 is inherently interoperable:

- It has a well-formed public definition.
- It is already in use in several important applications and industries.
- It complies with existing and developing standards (including image standards).
- It allows standards and structures to cross reference each other. The video data stream can contain structures defined by other standards, and vis a versa.
- It permits the same information to be interpreted in different ways within different domains without prejudice. To the video industry, the information is a continuous video stream. To the telecommunications industry, the same information is a sequence of bits to be transmitted. To the computer industry, that sequence of bits is interpreted as data structures.
- It permits independent formal definition of intellectual property protection, encryption, and other source coding descriptors by appropriately sanctioned expert groups (either ad hoc or extant).
- Experts groups and standards bodies can autonomously develop and

evolve specifications within domains of their expertise. The ASN.1 cross referencing capability achieves a degree of interoperation and coordination among parallel activities -- coding details are automatically resolved, avoiding redundant and/or conflicting efforts.

Cost/Performance Effectiveness --

- It is straightforward to recognize and decode with a single uniform procedure.
- Uniform decode hardware and software can be shared among industries and applications yielding economies of scale.
- Existing tools and expertise can reduce the time and cost of deployment.

Compactness --

- ASN.1 is compact, but not so much as to complicate decoding or to compromise other objectives. A typical ASN.1 header would be ~15 bytes for a 1 megabyte payload (i.e., ~.0014% overhead).
- A 7-byte ASN.1 header can be realized using the standard's indirect reference option. (In fact, a 2-byte context specific header may be realizable.) Given typical payload sizes, however, it is unlikely that this level of compactness will be required -- small payloads are either aggregated or infrequently used, especially in the video domain.

Rapid Capture --

- An ASN.1 header is signified by its first byte tag field (equal to 8) followed by length, object id, and payload. Thus, an ASN.1 header is straightforward to recognize in the data stream.
- ASN.1, like any length/identifier header, provides early identification of the payload.

Editability --

- ASN.1's structuring capabilities permit arbitrary editing, sequencing, structuring, and embedding of payload streams. All of this is accomplished within a single uniform mechanism, and without requiring unnecessary decoding of the payload itself.

APPENDIX A

TRANSPORT HEADER

A.0 Introduction

The header descriptor design provides for binding of a transport header to the main header. This is accomplished only by mandating that the transport header not be separated from the main header and its associated data which it is transporting. The function of the main header is to identify the data which follows. The function of the transport header is to help the main header and possibly its payload in its journey from origination to destination.

The following transport header design is "work in progress", and therefore is meant to be an example rather than the final structure. It supplements the discussion in Section 5.4. The design issues and interactions are a bit complex, so the principles of the transport header design are best illustrated by way of a correctly designed example. If adjustments are made to this design, then care must be taken concerning affects of such adjustments to the rest of the design and the functioning of the transport header. In particular, there is a rigid requirement that certain fields be pre-specified as to length, or as to length-specification (type fields) in their meaning.

It is necessary that the transport header be totally independent of any standards described by the main header. This is required because it may be necessary to change transport characteristics for all main headers on a given data stream, irrespective of the standards or formats of those main headers.

An example might be the need to provide improved error protection when moving a data stream from a high reliability fiber to a high error rate radio frequency transmission.

A.0.1 ASN.1 Transport Header Yet To Be Designed

The transport header design example illustrated here works together with the header/descriptor design as described in sections 1 through 5 of this SMPTE report.

No transport header has yet been designed for the ASN.1 syntax, illustrated in section 6. Thus, the ASN.1 syntax is, at present, only suitable for error-free channels which preserve the data and its order without contention from source to destination. In order for the ASN.1 syntax to meets its objectives of interoperability with imperfect or congested channels or media, a mechanism similar to the transport header example shown here will be required. It is hoped that a transport header design might be developed for the ASN.1 syntax system, possibly modelled on the transport header illustrated here, together with the main header/descriptor design. The enormous flexibility of ASN.1 syntax must be tempered to provide a limited number of options for transport headers, each with appropriate protection/correction mechanisms. It is hoped that registration rules and flexibility in ASN.1 can also be used to provide a

suitable format transport header design which is properly restricted. Byte alignment is also part of the structure of length fields. If bit alignment is needed for ASN.1, then further suitable adjustments will be required.

Another method to provide transport assistance is to convert from the main header/descriptor design of sections 1 through 5, to and from the ASN.1 notation. Since a transport header design is available for the main header/descriptor, conversion to this header would provide access to a transport header. When re-entering error and contention free environments, the header/descriptor could be re-converted to ASN.1 syntax.

If the ASN.1 method becomes popular, then it is hoped that a suitable ASN.1 transport mechanism might be developed.

A.1 Design objectives for the transport header

- The transport header must be standard-id independent, so that it can apply to all header standards equally and uniformly across the entire data stream.
- The transport header should be removable without damage to the function of the main header which it is helping to transport.
- A transport header must be able to be added to any header format or descriptor format without changing any of the meaning.
- The transport header formats should support "in the clear" protection of the main header and payload, where the bits are not altered, such that the transport header can be added and removed without any adjustment of bits within the main header, its descriptor and its payload.
- In addition to support for "in the clear" protection, more efficient protection should be supported (such as Reed Solomon), where the header, and possibly its payload are encoded.
- The transport header architecture should support one or more mechanisms for correcting burst errors.
- Optional support should be provided for encryption of the main header's descriptor, as well as optionally the data payload.
- The transport header should support authorization and use information, in helping the transport system determine which destinations are appropriate for further or final distribution.
- The transport header architecture should support backward play through the data stream.
- The transport header should support optional rapid header synchronization capture.
- The transport header should support communications networks by

providing information concerning the data's priority and value.

- The transport header should support timing reconstruction when utilizing networks which distort timing or ordering of data.

- The transport header should support data ordering requirements, when utilizing networks which might re-order the data.

- The transport header architecture should strike a balance between the opposing forces of flexibility and ease of use. Thus, a small number of options should be carefully chosen for maximum flexibility, with the small number of options allowing a simple interpretation. It is the fact of having a small number of options which allows easy interpretation.

- For transport functions which encode the header and/or its data, a simple "in the clear" length field should allow devices which cannot decode such data to skip to the next transport header.

- For devices which cannot process the transport header, a simple "in the clear" length field should allow such devices to skip directly to the header. Further, such devices should be able to easily interpret the transport header format such that they can quickly determine whether the main header is "in the clear", and therefore directly readable.

The following example design meets these objectives.

A.2 Three Types Of Transport Header Allocated In the Header Key

In order to meet these objectives, there are three types of transport headers. The first is the basic transport header, which provides the majority of transport capabilities. The second is the Redundancy Transport Header, which is used to protect against burst errors. The third is the Reverse Transport Header, which is used for reverse play.

Three of the two-byte header key's 256 possible codes are required for the transport headers. In the current 2-byte header key design, one byte is dedicated to error protection. The other byte is split into two 4 bit fields.

The first 4 bit field is the "length type" field. Codes 15 and 16 (numbers 14 and 15) are unallocated. These two codes enable the second 4 bit ID field to be used for special purposes such as designating the three transport header types. 32 such codes are available, leaving 29 codes unallocated if 3 codes are assigned to the three transport header types.

The basic transport header provides most of the capabilities which are required.

A.3 Functions Of The Transport Header

The functions of the transport header are as follows:

1. Sync reinforcement for those data transport media where it is desirable to rapidly or simply sync to the headers or header-data combinations on switching between streams.

2. Improved error protection via extra protection bits for both the transport header and the main header and its descriptor.

3. Conveyance of priority for the main header and its data, for those cases where a channel may be operating at capacity and thus where channel controllers must decide which headers and their payloads it must drop.

   Authorization keys may also be needed in order to verify priority. Network accessing methods, such as pricing-biding techniques, would also be supported here.

4. Encryption and Security information for the main header's descriptor, possibly combined with the descriptor's own encryption and security information, in order to protect the data stream following the main header.

   The protection optionally provided by the main header's descriptor may need to be augmented when the data stream is sent through public or vulnerable exposed channels.

5. Authorization information. Such information would indicate who could receive, who could edit and re-assemble with other material, etc. Also, copyright and royalty fee information would be enabled here. Perhaps automated mechanisms of fee for usage would be supported through this field.

6. Sequence numbers may be added where networks are used for transport which may reorder packets. In addition to sequence numbers, the combination of the transport header may require information from the main header's descriptor in order for the network to be able to guarantee delivery within known latencies and time windows. For out-of-order delivery, some networks can control the amount of time between a given delivery and the delivery of neighboring data. In the case of images and audio, some devices can accept out-of-order information in their buffers or processing units, but the time is constrained to within one or more frames, or fractions thereof.

7. Timing reconstruction. For those applications where exact timing relationships must be reconstructed from a mixed data stream, the transport header and the main header's descriptor would communicate to provide timing reconstruction information.

8. Reserved for future use.

9. Pad. There is a pad field at the end of the transport header in order to make the length of the transport header plus the main header, its descriptor, and optionally its payload come out to a length appropriate for

the error correction protection formats supported in item 2 (above).

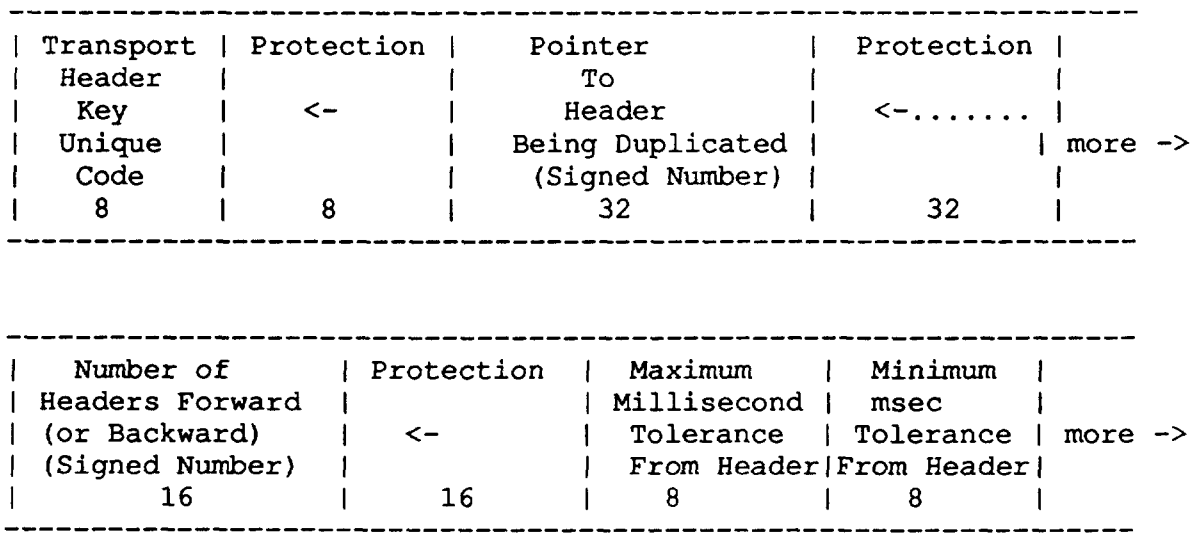
The construction of the transport header involves a strict ordering of fields as above, such that the sync reinforcement is always first, the error protection is always second, etc. In this way, if each field is present, its location is easily determined. The error protection field will always be in a known location, and a "scope of protection" within this field defines those fields which are protected in both the transport header as well as in the main header and its descriptor.

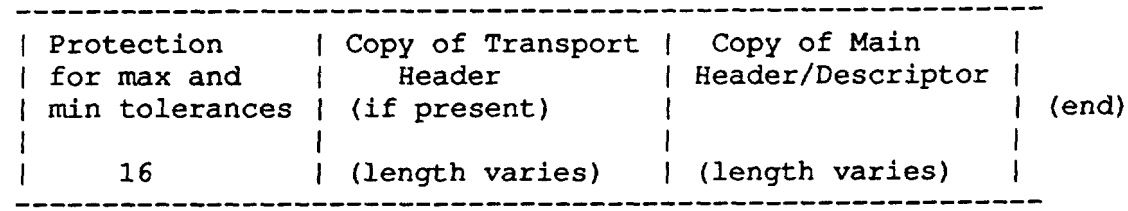
A.4 Redundancy Transport Header

The redundancy transport header provides a special function for error protection against burst errors on the data stream. Improved error protection against burst errors is achieved via redundant copies of future and previous headers (these transport headers providing redundancy will not be bound to a main header, and therefore represent an exception to the binding property of transport headers).

Such special redundancy transport headers will "stand alone" and be occasionally interspersed in the data stream. They will contain error-protected copies (via either the efficient or inefficient method) of some future or previous header, together with a pointer to that header, and a number indicating how many headers forward or backward will be traversed before reaching that header. The previous copies are useful for going backward through a data stream, and for reconstructing damaged data streams on physical media, such as disk.

Using a separate header key code, the redundancy transport header has the following format:





The transport header key contains a separate special code indicating that this is a redundancy transport header containing a copy of a future or previous header.

This field could possibly be followed by a length field, indicating how to skip past this duplicate header.

The pointer to the header being duplicated is analogous to a length field, but it points past several headers to the header being duplicated. It is a signed number so that it can reference previous as well as future headers. 32 bits of protection are provided.

A number of headers forward or backward is provided, indicating the location of the header being duplicated in number of headers rather than via a pointer (length).

The maximum and minimum milliseconds tolerance fields indicate the tolerances for separation times between this copy of a previous or future header, and the header itself. For channels which re-order, insert, and remove data, information about separation constraints is provided by these fields.

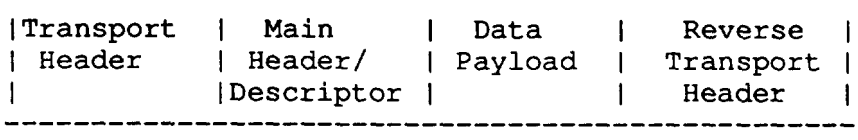
A copy of the transport header is provided, if there is a transport header on the header being copied.

Then finally there is a copy of the complete main header and its associated optional descriptor. None of the payload is duplicated.

A.5 Reverse Transport Header

In order to support reverse reading of the data stream, a reverse transport header can be utilized. The reverse transport header immediately follows the main header.

Thus the transport header is situated as follows:



If the main header is preceded by a transport header, then the reverse transport

header will point back to both the main header and the transport header. If the transport header is absent, then the reverse transport header will point back only to the main header, and the length backward to the transport header will be zero.

The reverse transport header format is as follows:

Reverse Transport Header Key Unique Code	Protection <-	Length Backward To Main Header	Protection <-	Length Backward To Transport Header	Protection <-
8	8	32	32	32	32

This design allows these reverse transport headers to be appended after the main headers to allow backward traversal through the data stream.

It should be noted that when redundant transport headers are in use to protect against burst errors, that reverse transport headers must follow each such redundant header. In that case, the length backward to the main header will be zero, and only the length backward to the transport header will have a non-zero value.

Note that this header has a fixed length. Thus, when encountering this header in the forward direction, no pointer to the next header in the form of a length field is required. The fixed length of 18 bytes is pre-determined, and can be used to skip to the next header. Also, there will never be a payload of data or any attachment to any headers in the forward direction.

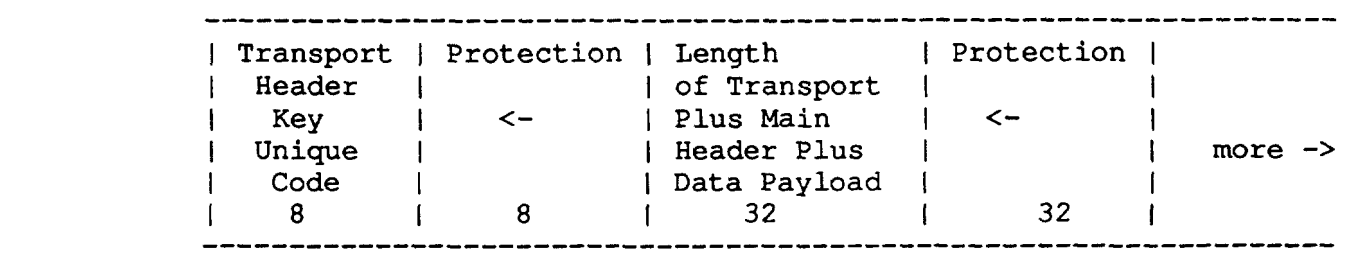
A.6 Transport Headers and Device Capture

It is important to remember that all devices which can edit the data stream must preserve the relationship of the pre-appended transport header to the main header. Also, the post-appended reverse transport header must also remain attached to the main header, if present. Thus, when a transport header is read, the following two headers must also be read before assuming the header and its data and transport have been passed.

When capturing a new data stream, it is necessary to read at least two headers to determine if the first main header is valid. If it had been preceded by a transport header which encoded the main header's descriptor and/or payload, then the data will not be readable without the transport header. Thus, capture is not achieved until the second header has been read, which would accomplish the determination of the first valid header, and its transport header, if present.

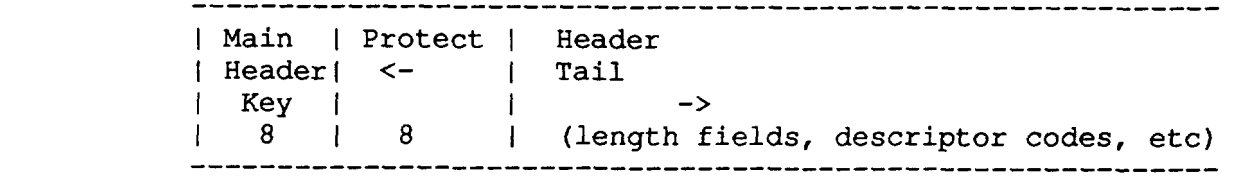
A.7 Transport Header Format

The transport header format is as follows:



(end of transport header ->)

Followed by a normal main header:



A.7.1 Header Code in the Header Key

The transport header begins with a normal header "key", consisting of one byte of key information and one byte of protection, as for the main header. However, the transport header uses a header key code reserved specifically for the transport header. The transport header tail differs from the main header tail, and has the format outlined above.

A.7.2 Length to Next Header after Main Header

The next field is a 32 bit length field, which points to the next header after the main header attached to this transport header. This length may be required to allow skipping past the main header and its data payload. This will be required by those devices which cannot provide error protection decoding, when the main header is protected using the types of error codes which scramble the main header. This length is protected by 32 additional protection bits to provide reasonably robust bit error correction, using the type of correction which does not scramble the 32 bits of length (e.g. Hamming Code).

A.7.3 Transport Header Length

This field indicates the length of the transport header. It therefore forms the mechanism to skip to the main header, if there is no desire to read any of the transport fields, and if the main header is not scrambled due to error protection, encryption, or other operations from the transport fields.

This field is also protected by a 32 bit field, using a non-scrambling error protection code.

A.7.4 Type Fields

There are eight 4-bit type fields, to allow 16 types for each of the eight fields in the transport header. These fields are (1) sync, (2) error protection and correction, (3) priority and authorization and bidding for priority, (4) authorization for data use, (5) encryption protection, (6) sequence number and out-of-order timing margins, (7) timing reconstruction information, and (8) a final field reserved for future use.

The 32 bits of type fields are protected by 32 bits of protection/correction code.

A.7.5 Sync

The first transport operation field is sync reinforcement. 16 types of codes are possible, each with a permanently assigned length. These types and lengths must be permanently assigned from the beginning. A possible set of 16 assignments for lengths might be:

Two types of sync codes could be used, with designed unique spectral

signatures. Each of the two types of codes could have one of the following lengths.

2, 4, 8, 16, 32, 64, 128, and 256 bytes

This would result in a total of 14 sync re-enforcement patterns. Sync type 0 would indicate an absence of the sync re-enforcement field. Sync type 15 could represent an additional special sync field, with a specified length.

It is necessary for all sync type lengths to be specified in advance. Although the codes for sync themselves can be specified later, they can only be specified once for each of the 15 valid sync types.

It should be noted that the transport header always begins exactly 26 bytes prior to the first byte of the sync-reinforcement field. Once the sync reinforcement field has been located, the transport header, and the main header have been located.

A.7.6 Error Protection

None of the fields previously described, which precede the error correction/protection field, will be protected by this field. However, all of the fields following and including this error protection/correction field, starting at the first bit, will be part of the error protection/correction group. The protection will extend through the rest of the transport header, and on into the main header and its descriptor, and for some of the formats into the payload as well.

Since sync re-enforcement is used to capture the data stream initially, it is probably not appropriate to error protect this sync. The sync codes are designed to be found within a stream. If protection is needed for the sync field, then special sync protection can be provided within the ample bytes available for sync codes.

Two types of error protection are supported. One type allows the protected data fields to be read, as is, leaving them "clear" but augmenting them with protection. This type is very inefficient. An example of this type of code is the Hamming code. The second type scrambles all of the bits into a robust code, such as Reed Solomon, for efficient protection. In the case of the efficient protection, the fields being protected will be completely unreadable without decoding. If the header format, the header length field, the descriptor type, and many other crucial fields are protected in this way, those devices which cannot decode the error correction would be unable to either read or skip the header. Thus, the transport header will contain a "length field", protected in the inefficient augmentation method rather than the efficient scrambling method, which will allow devices to find this length field and thereby skip the rest of the transport header and the entire main header, its descriptor, and its payload. A second "length of transport header" field will also be present and protected via the inefficient method.

A.7.6.1 Intentional Inflexibility

In order for all of the above mechanisms to operate properly and efficiently, it is necessary to limit the number of formats available in the transport header. Perhaps eight or sixteen types of each of the fields should be provided for, with all types being specified in advance. We are using sixteen as our example. Thus, there would be sixteen error correcting code formats and {_ code lengths, sixteen sync reinforcement field formats and lengths, sixteen priority fields and lengths, etc, with each field format having a specified length.

Since the error correction transport function is the most difficult with respect to format, a very limited number of field options will be provided under the scope of protection. Also, padding, which might be quite long, will be required at the end of the transport header before the main header in order to make the total length of the transport header plus the main header and its descriptor (and possibly payload) to be a simple-to-correct convenient known length corresponding to one of the sixteen protection types.

A.7.6.2 Possible Pre-Specified Protection Types

The pre-specified sixteen possible protection types might be:

Type 0 indicates that no protection/correction field is present.

Types 1 through 10 protect "in the clear", by inefficiently adding protection bits without scrambling, as in Hamming codes.

Types 1 through 5 protect the transport header, the main header, and its descriptor.

The types are as follows:

1. Protect all remaining transport header bits, beginning at the first bit of the error protection/correction field, all main header bits, and all descriptor bits. Do not protect any payload bits. The protection/correction bits are applied on every group of 64 bits. The total length of all fields being protected, excluding the error code bits themselves, must be a multiple of 64 bits. This is accomplished by the use of pad bits at the end of the transport header.

This implementation requires that memory be available to store the correction bits for the entire length of transport header, main header, and its descriptor.

For type 1, the total length of the protection code field is the total length over 16. Four bits for every 64 (68 bits total on 64 bits of data).

2. Same as 1, but total length over 8. Eight bits protecting every 64 (72 bits total for 64 bits of data).

3. Same as 1, but total length over 4. Sixteen bits protecting every 64 (80 bits total for 64 bits of data).

4. Same as 1, but total length over 3 (half as long as the fields being protected). 32 bits protecting every 64 (96 bits total for 64 data).

5. Same as 1, but total length over 2 (the same length as the fields being protected). 64 bits protecting every 64 (128 total for 64 data).

   Types 6 through 10 protect the payload in addition to the transport header, the main header, and its descriptor.

6. Same as type 1, except protect the payload as well. For type 6, the total length of the protection code field is the total length over 16. Four bits protecting every 64 (68 bits total 64 bits of data).

7. Same as 6, but total length over 8. Eight bits protecting every 64 (72 bits total for 64 bits of data).

8. Same as 6, but total length over 4. Sixteen bits protecting every 64 (80 bits total for 64 bits of data).

9. Same as 6, but total length over 3 (half as long as the fields being protected). 32 bits protecting every 64 (96 bits total for 64 data).

10. Same as 6, but total length over two (the same length as the fields being protected). 64 bits protecting every 64 (128 total for 64 data).

Types 11 through 15 protect by scrambling, as in Reed-Solomon coding, which is efficient.

Types 11 and 12 protect the transport header, the main header, and its descriptor.

11. Protect all remaining transport header bits, beginning at the first bit of the protection/correction field, all main header bits, and all descriptor bits. Do not protect any payload bits. The protection is applied using 16 bytes on every group of 144 bytes. The total length of all fields being protected, including the error code bits themselves, must be a multiple of 144 bytes. This is accomplished by the use of pad bits at the end of the transport header.

This implementation requires that memory be available to store the correction bytes for the entire length of transport header, main header, and its descriptor.

For type 11, the total length of the protection code is 16 bytes for every 144. Thus, there are 16 extra protection bytes in each 144 bytes being stored, resulting in 128 bytes of data after decoding.

12. Same as 11, but with 16 bytes protecting every 80 bytes, resulting in 64 bytes of data after decoding. The total of all lengths, beginning at the first bit of the protection/correction field, must be a multiple of 80 bytes.

Types 13 and 14 protect the payload in addition to the transport header, the main header, and its descriptor.

13. Same as 11, except the payload is also protected.

14. Same as 12, except the payload is also protected.

15. Same as 11, except 4 bytes protect 32 bytes (total of 36 bytes for 32 bytes of data). This format is for short header formats, and provides no payload protection.

A.7.6.3 Interleaving

In addition to the above mechanisms for error protection, some of the error protection formats can invoke interleaving. Interleaving can substantially reduce the problems associated with burst errors. Although the initial part of the transport header is subject to being "wiped out" by a burst error, presumably a copy of this section could be available previously in a redundancy transport header. Thus, once the protection format has been determined, then the rest of the transport header, beginning at the error protection field, plus the main header, its descriptor, if present, and optionally the data payload, can all be protected from burst errors via interleaving in addition to error correction methods.

Pre-defined interleaving methods can be incorporated with some of the types discussed above. Because interleaving is likely to involve as wide a spacing as is feasible, there will be a tradeoff between the length of the protected field, and natural multiples of the error protection sizes. The error protection group sizes for Hamming-type codes are much smaller than the error protection group sizes for Reed-Solomon-type codes. Interleaving must be some multiple larger again. Thus, useful interleaving may be restricted to longer lengths of fields being protected. One possibility is to have the interleaving spacing be the error protection group size divided into the total length.

However, this near-optimal format requires some complexity in unwinding the interleaving. For long protected fields, this may also involve a buffer which is the length of the field. Thus, there are potential issues to investigate with respect to how to universally and generally specify a powerful interleaving technique.

A.7.6.4 Error Detection

There is no provision for simple error detection in the above example types. Such detection could be provided via cyclical redundancy code (CRC), fire code, checksum, parity or other check method. Such may be useful in some cases. However, the focus on error correction is based on the need for headers to be interpreted without error in order to serve their function.

The descriptor in the main header can be used for detection codes for data payloads which should be checked, but which need not be corrected. This need not be standard-specific, since the descriptor can be standard independent.

Thus, error detection, as opposed to correction, is more appropriate in the descriptor than in the transport header.

A.7.6.5 Parameters Of Error Protection

The parameters of error detection shown in the above type examples need further investigation and refinement. The lengths and protection ratios proposed are known to be implementable in existing hardware, and are expected to be convenient in practices. However, further investigation of optimal parameters for error protection may help refine or revise the parameters suggested above.

A.7.7 Priority and Authorization or Bid for Priority

Finite bandwidth resources, such as satellite channels, long fiber channels, long real-time computer channels, terrestrial broadcast channels, and other channels with long distances cause long latency which naturally prevents error-retry. Thus, channel bandwidth allocation near saturation on real-time imagery streams takes the form of packet collisions. Such packets are most naturally the header/descriptor/payload combination, since each can have its own priority, and since each forms a constant priority grouping. The constant priority grouping would be the construction used by the originator.

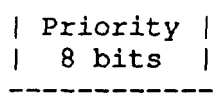
When sharing a finite-bandwidth channel, it may be necessary to pass some data and drop other data. In order for the channel's controlling device to fairly determine which packets to pass and which to drop, priorities for packets might be provided. In many spatial-frequency based compressed imagery formats such as DCT, Sub-band, and wavelets, the high frequencies represent tiny picture detail which might be lost without much picture degradation. However, the spatial low frequencies, the audio, and the motion vectors must be heavily protected, and may not be dropped without visible artifacts.

The type field would specify the length and format of the priority and/or authorization fields that follow. The length might have 2^type length (two to the power of the value in the type field, being 2, 4, 8, 16, 32, 64, 128, etc bytes). Type 0 still represents the absence of the priority field.

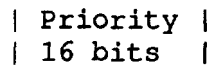
Since the priority and their authorization fields will compete at the highest level, it will be necessary for us to define their meaning at the outset. We will further need to define the mappings between the shorter and longer versions of each field.

The format of the fields might be as follows:

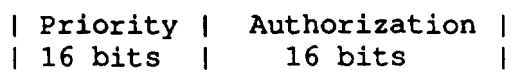
Type 0, 1 byte:



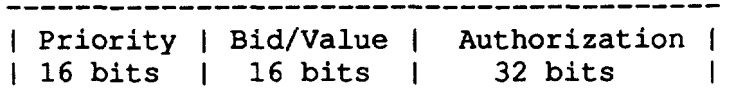
Type 1, 2 bytes:



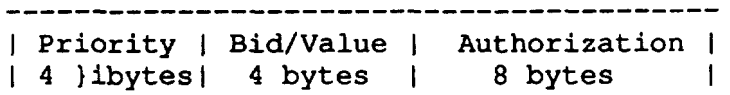
Type 2, 4 bytes:



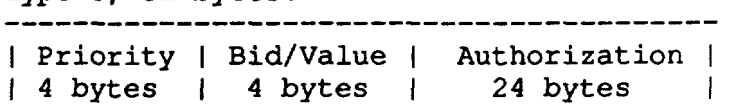
Type 3, 8 bytes:



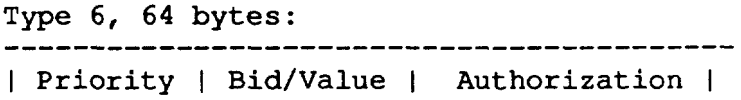
Type 4, 16 bytes:



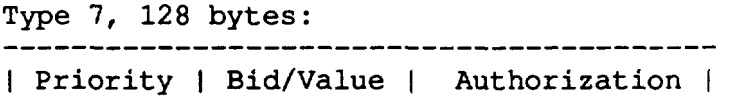
Type 5, 32 bytes:



Type 6, 64 bytes:



Type 7, 128 bytes:



etc.

The priority field varies from 1 to 8 bytes, allowing very detailed priority levels.

The Bid/Value field allows a packet to have a "bidding price" in a collision with other packets. Such a bidding price would be a value if the price had previously been accepted. A value would imply that tossing the packet would break a contract for delivery of the packet. The meanings of the Bid/Value fields would be tied directly to authorization codes, which would indicate the following:

1. Whether the header was authorized to bid.

2. The "credit rating" (or importance) of the bidder. This could potentially weight the priority field.

3. Whether the bid had been previously accepted, such that the Bid/Value field was the value paid for the payload's delivery. In such a case, tossing the packet would violate the contract. Presumably such a case would only occur when more contracts had been made than were available, such that packets were only tossed by other similar accepted-bid packets with an established value. This is a similar problem to "overbooking" on airlines.

4. Authorization may affect the bid/value. If commissions are paid on some bids or values, and not on others, the net bid or value may differ. This is similar to the problem of bids in different currencies, or direct bookings vs using agents. Thus authorization can indicate the source and/or type of a bid for these purposes.

The priority should be registered in entirety. Thus, the meaning of priority codes might be defined by registration. However, it may be desirable to have priorities have simple linear precedence order, with higher values representing higher priorities. One possible solution, is to define the first, or the first and second bytes of the priority to be linear magnitude precedence priority codes. Subsequent bytes, however, could be registered codes, with unique meanings which are standardized to help resolve priority conflicts.

Other than the potential interactions of authorization on priority and bid/value, authorization can have the following very important uses:

A.7.8 The Authorization Field

Uses of authorization

1. Pay-per view target encryption codes (in lock step with receiving system).

2. Channel authorization. For example, is a Satellite downlink channel signal authorized for use as a cable head-end?

3. Channel routing authorizations. For instance, are all authorized destinations only on network fork A, such that a source for sub-networks A and B need not carry the payload to sub-network B. This is the function of supporting a subset of all of the outputs involved in a "Y" connection.

4. End user authorization for teleconferencing, to indicate who can be included in the teleconference, including who may observe and who may originate.

5. Privacy and protection against unauthorized acceptance or origination of the signal in any use. For example, protection against real-time datastream "hackers", or against unauthorized video-phone "wire tapping".

6. Authorized user enablement codes. Such codes would authorize user systems for future authorized codes. For example, when a cable subscriber adds a new channel, an enablement code would be sent to their decoder to add authorization interpretation and viewing for the new channel.

7. Diagnostic, statistic, and rating codes for exploring network loads, active users, show ratings, unintentional network disconnects, channel error rates, etc.

8. Copyright information indicating ownership.

9. Copyright fee structures, including where to pay fees.

10. Indications of who may edit a work, and whether it may be included in other works, and fee structures for doing so.

11. Possibly an automated mechanism could be constructed to automatically negotiate rights based on pre-arranged "willing to pay" algorithms, so that clips can be included without undue complication.

A.7.9 Encryption

One function of the transport header is to provide one or more encryption keys for deciphering the payload and descriptor.

Many protected users may wish to protect against unauthorized deciphering of the descriptor, since it may contain valuable information which could help in deciphering the payload. Codes could be used for encryption keys, for example, to unlock descriptors. Descriptors, in turn, may contain more elaborate encryption codes to further unlock the payload.

Based upon successful authorization code interactions, encryption codes can be deciphered and applied against the descriptor, the payload, or both.

As usual, a type 0 represents that no encryption field is present. 15 types are available, with 15 pre-specified associated lengths, for encryption.

Although the lengths must be prespecified, the meaning of the encryption, or its associated technique, can be completely private.

Complex encryption algorithms can be developed and updated between imbedded codes in the receiving device, codes in the descriptor, and possibly algorithms transmitted and updated via descriptors.

A.7.10 Sequence Numbers

The sequence numbering field specifies not only packet ordering, but also windows of order and groupings. For example, in some systems various headers and their associated payloads form packets which can update the screen in any order during the frame time before the buffer switch for viewing. However, motion vectors might need to precede compressed image deltas. Thus not only packet grouping, but packet general ordering might be specified.

On some networks, lower priority packets are delayed, rather than dropped. In such networks, it is necessary for the network controlling mechanisms to understand the bounds on acceptable delay for packets and groups of packets.

This field contains a series of codes for defining tolerance of imagery and other real-time streams for being received out-of-order.

Type 0 means no sequence field is present. 15 valid type fields with the associated pre-specified lengths are available.

A.7.11 Timing Reconstruction

In real-time data streams, it is often necessary to reconstruct precise timing after this timing is disrupted during transport. Timing reconstruction information, concerning the times at which events should occur, are specified in this field. Times can be specified as absolute times, where the transport delays and their bounds are known. Relative times can be specified relative to an arbitrary "start of real-time stream" clock marker which is set by the receiving device upon receiving the first displayable buffer load.

Synchronization between audio and image, between multiple audio streams, or between streams from multiple sources, is handled via the timing reconstruction field.

Re-synchronization for removing cumulative jitter effects can also be enabled through the use of this field.

A type of 0 indicates an absence of the timing reconstruction field. The fifteen available codes will have prespecified lengths, although their timing meanings may be deferred from some of the types. Of course, each of the types can only receive a single meaning, which meaning must stay in place from then on. The lengths for such unspecified codes must all be specified in advance, however.

A.7.12 Reserved For the Future

This field is unspecified in content and length. Because the total length of the transport header is known, and because this is the last field in the header prior

to the pad, this field{_ can maintain flexibility for future use by remaining completely unspecified.

All other fields must at least have their lengths specified for each type value.

A.7.13 Pad Bits

Pad Bits make the lengths simple for error correction processing. This is accomplished by making the total of the error correction/protection field range be the appropriate length for the error correction format being used. For example, using a 64-bit length type, the length after the error protection/correction field must be a multiple of 64 bits. Thus if the scope of the protection includes additional transport fields, such as priority and encryption, plus a header and its descriptor, the pad bits would make the sum a proper multiple of 64 bits.

APPENDIX B

ILLUSTRATIVE EXAMPLES OF HEADER DECODING USING "C"

B.0 Background

It is often instructive to represent a design as a computer program written in some appropriate language (in this case C). It verifies the design and provides a basis for comparing the cost and performance of design alternatives. The C language was chosen because it is reasonably universal.

Conciseness and consistency are foremost considerations in enabling comprehension and fair comparison; optimal performance is of secondary importance. Optimizations and enhancements would be added in preparation for commercial distribution.

Two programs are described.: One decodes a compact header and one decodesan ASN.1 header. They are similar in appearance and use the same basic steps. The primary difference is the compact header decoder selects between multiple formats using table lookups, while an ASN.1 header has only one extensible format.

Each program extracts the block length and standard identifier from the header, and then calls a corresponding function to process the payload.

B.1 Compact Header Decoder

The following program decodes a packet with a compact header. If the packet format is predefined, it calls the corresponding predefined function, otherwise it extracts the standard identifier and block length in a manner similar to the algorithm described in Section 5.2.8. It uses the identifier to lookup a decoding function (f), and ignores any blocks with unknown identifiers.

Two table lookups are used to decode the compact header. The length-type table (lt table) contains information used to decode predefined messages and block length. The identifier table (id table) contains information used to decode the standard identifier.

One obvious optimization is to combine the two table lookups into a single 256 entry lookup. This reduces the instruction path for some cases, but increases memory requirements.

The identifier is left as a string of bytes used to compute a hash table lookup of a decoding function. If a sovereign state field exists, it is processed together with the standard identifier, but a separate hash table is used. The hash table lookup is performed by the procedure lookup() which takes identifier address,

identifier length, and table selection as arguments, and returns a pointer to the corresponding function (f).

B.1.1 Cautionary Notes

Certain header formats are not yet defined or are reserved for future use. The program below does not support these formats.

Predefined message types are not yet standardized. To make the code complete a dummy function call fake() has been used. When the functions are standardized, the lt table would change accordingly.

Block length is assumed to fit within one 32 bit word. Extending the program and/or the C language to support larger word sizes, thus larger block lengths, is possible and likely to happen as 64 bit processor architectures emerge.

Bit field ordering and assignment are not yet defined. Choices made in the program below will require further consideration in the context of standardization.

If an unknown identifier is encountered, the lookup function will return a pointer to an appropriate default function that ignores the payload and displays an informative message.

B.1.2 Program Text

The program has two parts---the first part contains table and procedure declarations, the second part (at the end) contains the dozen or so statements actually executed. Throughout the code descriptive notes (comments that are not executed) are placed between comment delimiters (/*...*/).

```
/* Compact header has one of two forms:
 *
 * Each character in the strings below represents a byte; bytes between
 * square brackets are optional; payload bytes are not counted
 *
 * 2 byte (minimum) for predefined messages:
 *
 *   "ke[p...p]"
 *
 * Extended header for longer blocks:
 *
 *   "kel[...l][e...e]i[...i]"
 *
 * Key:
 *
 *   k :: key byte (length type and id type, presence of a readable description)
 *   e :: error byte
 *   l :: length byte
 *   i :: id byte
```